Pocket Linux Guide

David Horton

<dhorton@megsinet.net.NOSPAM>

Revision History

Revision 1.2	2003−05−31	Revised by: DH
Corrected errors in "strip −o *library*" commands.		
Revision 1.1	2003−05−21	Revised by: DH
Bug fixes, typo corrections and improved XML markup.		
Revision 1.0	2003−02−17	Revised by: DH
Initial Release, reviewed by LDP.		

The Pocket Linux Guide is for anyone interested in learning the techniques of building a GNU/Linux system from source code. The guide is structured as a project that builds a small diskette−based GNU/Linux system called Pocket Linux. Each chapter explores a small piece of the overall system explaining how it works, why it is needed and how to build it. After completing the Pocket Linux project, readers should possess an enhanced knowledge of what makes GNU/Linux systems work as well as the confidence to explore larger, more complex source−code−only projects.

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Legal Information

1. Copyright and License

2. Disclaimer

No liability for the contents of this document can be accepted. Use the concepts, examples and information at your own risk. There may be errors and inaccuracies, that could be damaging to your system. Proceed with caution, and although this is highly unlikely, the author(s) do not take any responsibility.

All copyrights are held by their by their respective owners, unless specifically noted otherwise. Use of a term in this document should not be regarded as affecting the validity of any trademark or service mark. Naming of particular products or brands should not be seen as endorsements.

Introduction

1. About Pocket Linux

The Pocket Linux Guide demonstrates how to build a small console−based GNU/Linux system using only source code and a couple of diskettes. It is intended for Linux users who would like to gain a deeper understanding about how their system works beneath the shroud of distribution specific features and tools.

2. Prerequisite Skills

This guide is intended for intermediate to advanced Linux users. It is not intentionally obscure, but certain assumptions about the readers skill level are made. Success with this guide depends in part on being able to perform the following tasks:

- Use basic shell commands
- Reference man and info pages
- Build a custom Linux kernel
- Compile source code using make and related tools

3. Project Format

The Pocket Linux Guide takes a hands−on approach to learning. The guide is written with each chapter building a piece of an overall project. Chapters are further broken into sections of Analysis, Design, Construction and Implementation. This format is derived from Rapid Application Development (RAD) methodology. Without going into detail about design methodologies, the sections may be summed up as follows.

- The Analysis section gives a high−level overview of what is to be accomplished in each chapter. It will introduce the tasks that need to be completed and why they are important to the overall system.
- The Design section defines the source code packages, files and configuration necessary to address the requirements set forth in the Analysis section. Much of the theory of why certain system files exist and what their purpose is can be found here.
- The Construction section is where all the hands on action takes place. This section goes into detail about building source code and configuring the system files.
- The Implementation section will test the proper operation of the project at the end of each chapter. Often there are a few shell commands to perform and samples of expected screen outputs are given.

Readers interested in learning more about RAD may want to consult a textbook covering systems analysis and design or visit the following University of California, Davis website on the subject: http://sysdev.ucdavis.edu/WEBADM/document/rad−stages.htm.

4. Help & Support

Readers are encouraged to visit the Pocket Linux Resource Site at http://my.core.com/~dhorton/linux/pocket/. The resource site is home to:

- information about the Pocket Linux mailing list.

- a collection of diskette images for various chapters.
- a list of additions and corrections to be addressed in upcoming versions.

5. Feedback

For questions and comments about Pocket Linux please visit the resource site and subscribe to the mailing list.

Chapter 1. Project Initiation

1.1. A Brief History of GNU/Linux

In the early 90's GNU/Linux systems consisted of little more than a beta−quality Linux kernel and a small collection of software ported from the GNU project. It was a true hacker's operating system. There were no CD−ROM's or GUI installation tools; everything had to be compiled and configured by the end user. Being a Linux Expert meant knowing your system inside and out.

Toward the middle of the decade several GNU/Linux distributions began appearing. One of the first was Slackware in 1993 and since then there have been many others. Even though there are many "flavors" of Linux today, the main purpose of the distribution remains the same. The distribution automates many of the tasks involved in GNU/Linux installation and configuration taking the burden off of the system administrator. Being a Linux Expert now means knowing which button to click in the GUI administration tool.

Recently there has been a yearn for a return to the "good old days" of Linux when men were men, sysadmins were hardcore geeks and everything was compiled from source code. A notable indication of this movement was the publication of the Linux From Scratch HOWTO version 1.0 by Gerard Beekmans in 1999. Being a Linux Expert once again means knowing how to do it yourself.

For more historical information, see Ragib Hasan's "History of Linux" at http://ragib.hypermart.net/linux/

1.2. The Goal of Pocket Linux

The purpose of Pocket Linux is to support and encourage people who wish to build a GNU/Linux system from nothing but source code. It is not intended to be a full featured system, but rather to give the reader a taste of what is involved in building an operating system from source code. After completing the Pocket Linux system the reader should have enough knowledge to confidently build almost any project using only source code. Given this direction we can put a few constraints on the project.

- The main focus should be learning. The project should not just describe how to do something, it should also describe why it should be done.
- The required time commitment should be minimal and manageable.
- The project should not require any investment in additional hardware or reconfiguration of existing hardware to set up a lab environment.
- Readers should not need to know any programming languages in order to complete the project.
- To remain true to the spirit of GNU/Linux, all software used in the project should be covered under the GNU/GPL or another, similarly liberal, open−source license.

1.3. Working Within The Constraints

The Pocket Linux project gets its name from the fact that the bulk of the project fits onto two diskettes making it possible to carry the entire, working system around in one's pocket. This has the advantage of not requiring any additional hardware since any PC can be booted from the diskettes without disrupting any OS that exists on the hard drive. Using diskettes also partially addresses the aspect of time commitment, because the project size and complexity is necessarily limited by the 1.44 Megabyte size of the installation media.

To further reduce the time commitment, the Pocket Linux project is divided into several phases, each one chapter in length. Each phase builds only a small piece of the overall project, but at the same time the conclusion of each chapter results in a self−contained, working system. This step−by−step approach should allow readers to pace themselves and not feel the need to rush to see results.

Chapters are further subdivided in to four sections. The first two sections, analysis and design, focus on the theory of what is to be accomplished in each phase and why. The last two sections, construction and implementation, detail the steps needed to do the actual building. Advanced readers, who may be familiar with the theories laid out in a particular chapter are encouraged to gloss over the analysis and design sections in the interest of time. The separation of theory from hands−on exercises should allow readers of all skill levels to complete the project without feeling either completely lost or mired in too much detail.

Finally, the Pocket Linux project will strive to use GNU/GPL software when possible and other open−source licensed software when there is no GNU/GPL alternative. Also, Pocket Linux will never require any programming more complex than a BASH shell script.

Chapter 2. A Simple Prototype

2.1. Analysis

Since this is the first phase of the project it will be kept very simple. The goal here is not to create the ultimate GNU/Linux system on the first try. Instead, we will be building a very minimal, working system to be used as a building block in subsequent phases of the project. Keeping this in mind, we can list a few goals for phase one.

- Keep it simple to avoid stressing out.
- Build something that works for instant gratification.
- Make something that it is useful in later phases of the project.

2.2. Design

2.2.1. Simplification

Take a moment to skim through the Bootdisk−HOWTO or the From−PowerUp−to−BASH−Prompt−HOWTO. These HOWTO documents can be found online at http://www.tldp.org/docs.html#howto. Both documents offer an excellent view of what it takes to get a GNU/Linux system up and running. There is also a lot of information to digest. Remember that one of our goals is, "keep it simple to avoid stressing out," so we want to ignore everything but the absolutely critical pieces of a boot / root diskset.

Basically it boils down to the following required items:

- A boot loader
- The Linux kernel
- A shell
- Some `/dev` files

We don't even need an init daemon. The kernel can be told to run the shell directly by passing it an option through the boot loader.

For easy construction we will build a two−disk boot / root set rather than trying to get everything onto a single diskette. The boot loader and kernel will go on the boot disk and the shell will reside on the root disk. Both disks will need device files to function properly.

2.2.2. Boot Disk

For the boot disk we'll want to use a kernel that does not require modules for the hardware we need to access. Mainly, it should have compiled−in support for the floppy drive, ram disk and a text−based console. If such a kernel is not available, it will need to be built from source code using the Kernel−HOWTO as a guide. Once the kernel is ready we can copy it to a diskette that has been prepared with a filesystem (a.k.a. formatted). The diskette will need a few `/dev` files and a `lilo.conf` file in order to get LILO installed. The Bootdisk−HOWTO and the lilo.conf(5) manpage will be helpful in designing the `lilo.conf` file.

2.2.3. Root Disk

For the root disk we will need a floppy that has been prepared with a filesystem. We will also need a BASH shell that is statically linked so we can avoid the additional complexities of shared libraries. The **configure** program in the BASH source code recognizes the `--enable-static-link` option for this feature. We will also be using the `--enable-minimal-config` option to keep the BASH binary down to a manageable size. Additional requirements for the root disk are a `/dev` directory and a device file for the console. The `console` device is required for BASH to be able to communicate with the keyboard and video display.

2.2.4. CPU Compatibility

There is one other, less obvious requirement to keep in mind and that is CPU compatibility. Each generation of CPU features a more complex architechture than its predecessor. Late generation chips have additional registers and instructions when compared to an older 486 or 386. So a kernel optimized for a new, fast 6x86 machine will not run on an older boxes. (See the `README` file in the Linux kernel source code for details.) A BASH shell built for a 6x86 will probably not run on an older processor either. To avoid this problem, we can choose the 386 as a lowest common denominator CPU and build all the code for that architecture.

2.3. Construction

In this section, we will be building the actual boot disk and root disk floppies. Lines starting with `bash#` indicate a shell command.

2.3.1. Prepare the boot disk floppy

Insert a blank diskette labeled "boot disk".

```
bash# mke2fs -m0 /dev/fd0
bash# mount /dev/fd0 /mnt
```

2.3.2. Build the kernel

```
bash# cd /usr/src/linux
bash# make menuconfig
```

Be sure to configure support for the following:

- 386 processor
- Floppy disk
- RAM disk
- Console on virtual terminal

```
bash# make dep
bash# make clean
bash# make bzImage
```

2.3.3. Copy the kernel to diskette

```
bash# mkdir /mnt/boot
bash# cp /usr/src/linux/arch/i386/boot/bzImage /mnt/boot/vmlinuz
```

2.3.4. Copy the LILO boot loader

```
bash# cp /boot/boot.b /mnt/boot/boot.b
```

2.3.5. Create device files that LILO needs

```
bash# mkdir /mnt/dev
bash# cd /mnt/dev
bash# mknod fd0 b 2 0
bash# mknod console c 5 1
```

2.3.6. Write a simple lilo.conf

```
bash# mkdir /mnt/etc
bash# cd /mnt/etc
```

Use an editor like vi, emacs or pico to create the following `lilo.conf` file:

```
# /etc/lilo.conf − boot loader configuration file
#
boot=/dev/fd0
compact
prompt
read−only
vga=normal
image=/boot/vmlinuz
label=bootdisk
append="load_ramdisk=1 prompt_ramdisk=1"
root=/dev/fd0
#
# end of /etc/lilo.conf
```

2.3.7. Install the LILO boot loader

```
bash# lilo −r /mnt
```

2.3.8. Unmount the boot disk

```
bash# cd /
bash# umount /mnt
bash# sync
```

2.3.9. Prepare the root disk floppy

Insert a blank diskette labeled "root disk".

```
bash# mke2fs −m0 /dev/fd0
bash# mount /dev/fd0 /mnt
```

2.3.10. Build BASH

Get the bash−2.05 source code package from ftp://ftp.gnu.org/gnu/bash/ and untar it into the `/usr/src` directory.

BASH version 2.05b, the latest version at the time of this writing, will not build successfully when using the `−−enable−minimal−config` option. This leaves two choices. We can either fix 2.05b by applying the patch posted on gnu.bash.bug under the subject, "Compile error in execute_cmd.c with −−enable−minimal−config" or we can simply use the 2.05a version.

```
bash# cd /usr/src/bash−2.05a
bash# ./configure −−enable−static−link \
  −−enable−minimal−config −−host=i386−pc−linux−gnu
bash# make
bash# strip bash
```

2.3.11. Copy BASH to the root disk

```
bash# mkdir /mnt/bin
bash# cp bash /mnt/bin/bash
bash# ln −s bash /mnt/bin/sh
```

2.3.12. Create device files that BASH needs

```
bash# mkdir /mnt/dev
bash# mknod /mnt/dev/console c 5 1
```

2.3.13. Unmount the root disk

```
bash# cd /
bash# umount /mnt
bash# sync
```

2.4. Implementation

2.4.1. System startup

Follow these steps to boot the system:

- Restart the PC with the boot disk in the floppy drive.
- When the LILO prompt appears, type **`bootdisk init=/bin/sh`** and press **Enter**.
- Insert the root disk when prompted.

If all goes well the screen should look something like the example shown below.

```
boot: bootdisk init=/bin/sh
Loading bootdisk
Uncompressing Linux... Ok, booting kernel.
..
.. [various kernel messages]
..
VFS: Insert root floppy disk to be loaded into RAM disk and press ENTER
RAMDISK: ext2 filesystem found at block 0
RAMDISK: Loading 1440 blocks [1 disk] into ram disk... done.
VFS: Mounted root (ext2 filesystem) readonly.
Freeing unused kernel memory: 178k freed
# _
```

2.4.2. Testing what works

Try out a few of BASH's built−in commands to see if things are working properly.

```
bash# echo "Hello World"
bash# cd /
bash# pwd
bash# echo *
```

2.4.3. Noting what does not work

Try out a few other familiar commands.

```
bash# ls /var
bash# mkdir /var/tmp
```

Notice that only commands internal to BASH actually work and that external commands like **ls** and **mkdir** do not work at all. This shortcoming is something that can be addressed in a future phase of the project. For now we should just enjoy the fact that our prototype boot / root diskset works and that it was not all that hard to build.

2.4.4. System shutdown

Remove the diskette from fd0 and restart the system using **CTRL−ALT−DELETE**.

Chapter 3. Saving Space

3.1. Analysis

One of the drawbacks in the prototype phase of the project was that the diskset was not all that useful. The only commands that worked were the ones built into the BASH shell. We could improve our root disk by installing commands like **cat**, **ls**, **mv**, **rm** and so on. Unfortunately, we are short on space. The current root disk has no shared libraries so each utility would have to be statically−linked just like the BASH shell. A lot of big binaries together with a static shell will rapidly exceed the tiny 1.44M of available disk space. So our main goal in this phase should be to maximize space savings on the root disk and pave the way for expanded functionality in the next phase.

3.2. Design

Take another look at the Bootdisk−HOWTO and notice how many utilities can be squeezed onto a 1.44M floppy. There are three things that make this possible. One is the use of shared libraries. The second is stripped binaries. And the third is the use of a compressed filesystem. We can use all of these techniques to save space on our root disk.

3.2.1. Shared Libraries

First, in order to use shared libraries we will need to rebuild the BASH shell. This time we will configure it without using the `--enable-static-link` option. Once BASH is rebuilt we need to figure out which libraries it is linked with and be sure to include them on the root disk. The **ldd** command makes this job easy. By typing **ldd bash** on the command−line we can see a list of all the shared libraries that BASH uses. As long as all these libraries are copied to the root disk, the new BASH build should work fine.

3.2.2. Stripped Binaries

Next, we should strip any binaries that get copied to the root disk. The manpage for **strip** does not give much description of what it does other than to say, "strip discards all symbols from the object files." It seems like removing pieces of a binary would render it useless, but this is not the case. The reason it works is because a large number of these discarded symbols are used for debugging. While debugging symbols are very helpful to programmers working to improve the code, they do not do much for the average end−user other than take up more disk space. And since space is at a premium, we should definitely remove as many symbols as possible from BASH and any other binaries before we copy over them to the ramdisk.

The process of stripping files to save space also works with shared library files. But when stripping libraries it is important to use the `--strip-unneeded` option so as not to break them. Using `--strip-unneeded` shrinks the file size, but leaves the symbols needed for relocation intact which is something that shared libraries need to function properly.

3.2.3. Compressed Root Filesystem

Finally, we can tackle the problem of how to build a compressed root filesystem. The Bootdisk−HOWTO suggests three ways of constructing a compressed root filesystem using either a ramdisk, a spare hard drive partition or a loopback device. This project will concentrate on using the ramdisk approach. It seems logical

that if the root filesystem is going to be run from a ramdisk, it may as well be built on a ramdisk. All we have to do is create a second extended filesystem on a ramdisk device, mount it and copy files to it. Once the filesystem is populated with all the files that the root disk needs, we simply unmount it, compress it and write it out to floppy.

For this to work, we need to make sure the kernel is configured with ramdisk support and a default size of 4,096K. If the ramdisk size is something other than 4096K this can be fixed by adding the line "ramdisk=4096" to the development system's `lilo.conf` file. The lilo.conf(5) man page provides additional information.

3.3. Construction

This section is written using ramdisk seven (`/dev/ram7`) to build the root image. There is nothing particularly special about ramdisk seven and it is possible to use any of the other available ramdisks provided they are not already in use.

3.3.1. Create a ramdisk

```
bash# dd if=/dev/zero of=/dev/ram7 bs=1k count=4096
bash# mke2fs -m0 /dev/ram7
bash# mount /dev/ram7 /mnt
```

3.3.2. Rebuild the BASH shell

```
bash# cd /usr/src/bash-2.05a
bash# make distclean
bash# ./configure --enable-minimal-config --host=i386-pc-linux-gnu
bash# make
bash# strip bash
```

3.3.3. Determine which libraries are required

```
bash# ldd bash
```

Note the output from the **ldd** command. It should look similar to the example below.

```
bash# ldd bash
  libdl.so.2 => /lib/libdl.so.2 (0x4001d000)
  libc.so.6 => /lib/libc.so.6 (0x40020000)
  /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

3.3.4. Copy BASH and its libraries to the ramdisk

```
bash# mkdir /mnt/bin
bash# cp bash /mnt/bin
bash# ln -s bash /mnt/bin/sh
bash# mkdir /mnt/lib
bash# strip --strip-unneeded -o /mnt/lib/libdl.so.2 /lib/libdl.so.2
bash# strip --strip-unneeded -o /mnt/lib/libc.so.6 /lib/libc.so.6
bash# strip --strip-unneeded -o /mnt/lib/ld-linux.so.2 /lib/ld-linux.so.2
bash# chmod +x /mnt/lib/*
```

Using **strip −o** might seem an odd way to copy library files from the development system to the ramdisk. What it does is strip the symbols while the file is in transit from the source location to the destination. This has the effect of stripping symbols from the library on the ramdisk without altering the libraries on the development system. Unfortunately file permissions are lost when copying libraries this way which is why the **chmod +x** command is then used to set the execute flag on all of the libraries on the rootdisk.

3.3.5. Create a console device

```
bash# mkdir /mnt/dev
bash# mknod /mnt/dev/console c 5 1
```

3.3.6. Compress the ramdisk image

```
bash# cd /
bash# umount /dev/ram7
bash# sync
bash# dd if=/dev/ram7 of=~/phase2-image bs=1k
bash# gzip -9 ~/phase2-image
```

3.3.7. Copy the compressed image to diskette

Insert the floppy labeled "root disk" into drive fd0.

```
bash# dd if=~/phase2-image.gz of=/dev/fd0 bs=1k
```

3.4. Implementation

3.4.1. System startup

Follow these steps to boot:

- Restart the PC using the lilo boot disk from the previous chapter.
- At the LILO prompt, type **`bootdisk init=/bin/sh`** and press **Enter**.
- Insert the new, compressed root disk when prompted.

The screen output should be similar to the following example:

```
boot: bootdisk init=/bin/sh
Loading bootdisk
Uncompressing Linux... Ok, booting kernel.
..
.. [various kernel messages]
..
VFS: Insert root floppy to be loaded into RAM disk and press ENTER
RAMDISK: Compressed image found at block 0
VFS: Mounted root (ext2 filesystem) read-write.
Freeing unused kernel memory: 178k freed
# _
```

3.4.2. Verify results

If the implementation was successful, this new root disk should behave exactly like the root disk from the previous chapter. The key difference is that this compressed root disk has much more room to grow and we will put this extra space to good use in the next phase of the project.

3.4.3. System shutdown

Remove the diskette from fd0 and restart the system using **CTRL−ALT−DELETE**.

Chapter 4. Some Basic Utilities

4.1. Analysis

In the previous chapter it might seem like we did not accomplish very much. A lot of energy was expended redesigning the root disk, but the functionality is basically the same as in the initial prototype phase. The root disk still does not do very much. But we did make significant improvements when it comes to space savings. In this chapter we will put that extra space to good use and start cramming the root disk with as many utilities as it can hold.

The first two root disks we built only had shell built−in commands like **echo** and **pwd**. This time it would be nice to have some of the commonly used external commands like **cat**, **ls**, **mkdir**, **rm** and such on the root disk. Keeping this in mind we can define the goals for this phase as follows:

- Retain all of the functionality from the previous root disk.
- Add some of the commonly used external commands.

4.2. Design

4.2.1. Determining Required Commands

The first question that might come to mind is, "How do we know which commands are needed?" It is possible to just start with **cat** and **ls** then install other commands as we discover a need for them. But this is terribly inefficient. We need a plan or a blueprint to work from. For this we can turn to the Filesystem Hierarchy Standard (FHS) available from http://www.pathname.com/fhs/. The FHS dictates which commands should be present on a GNU/Linux system and where they should be placed in the directory structure.

4.2.2. Locating Source Code

The next logical question is, "Now that we know what we need, where do we get the source code?" The answer to this question can be found by searching the Internet. There are several good Internet resources out there that can aid us in our quest for source code. One good place to start looking is the Linux Software Map (LSM) on Ibiblio. Ibiblio's LSM search page can be found by navigating to http://www.ibiblio.org/pub/linux. By using the names of commands as keywords, we should be able to come up with the name and location of the corresponding source code package.

4.2.3. Leveraging FHS

So let's look at the FHS requirements for the `/bin` directory. The first few commands in the list are **cat**, **chgrp**, **chmod** & **chown**. Using these as keywords in an LSM search we discover that we need GNU's textutils package for cat and GNU's fileutils package for **chmod**, **chgrp** & **chown**. In fact quite a few of the commands in `/bin` come from GNU's fileutils. So a good way to tackle the problem of finding source code might be to group the commands together by package as shown below.

- The BASH shell −− **echo**, **false**, **pwd**, **sh**, **true**
- GNU textutils −− **cat**
- GNU fileutils −− **chgrp**, **chmod**, **chown**, **cp**, **dd**, **df**, **ln**, **ls**, **mkdir**, **mknod**, **mv**, **rm**, **rmdir**, **sync**
- GNU sh−utils −− **date**, **hostname**, **stty**, **su**, **uname**

These four packages do not contain all of the commands in the `/bin` directory, but they do represent of over 70% of them. That should be enough to accomplish our goal of adding some of the commonly used external commands. We can worry about the remaining commands in later phases of the project.

4.3. Construction

Rather than copying files directly to the ramdisk, we can make things easier by setting up a staging area. The staging area will give us room to work without worrying about the space constraints of the ramdisk. It will also provide a way to save our work and make it easier to enhance the rootdisk in later phases of the project.

The staging procedure will work like this:

1. Create a directory structure as defined in the FHS.
2. Copy in the files from phase 2's root disk.
3. Build the four new packages from source code.
4. Install files into the correct FHS directories.
5. Strip the binaries to save space.
6. Check library dependencies.
7. Copy to the whole directory structure to the ramdisk.
8. Compress the ramdisk and write it out to floppy.

4.3.1. Create a staging area

```
bash# mkdir ~/staging
bash# cd ~/staging
bash# mkdir bin boot dev etc home lib mnt opt proc root sbin tmp usr var
bash# mkdir var/log var/run
```

4.3.2. Copy contents of phase 2 rootdisk

```
bash# dd if=~/phase2-image.gz | gunzip -c > /dev/ram7
bash# mount /dev/ram7 /mnt
bash# cp -dpR /mnt/* ~/staging
bash# umount /dev/ram7
bash# rmdir ~/staging/lost+found
```

4.3.3. Install "cat" from GNU Textutils

```
bash# cd /usr/src/textutils-2.1
bash# ./configure --host=i386-pc-linux-gnu
bash# make
bash# cd src
bash# cp cat ~/staging/bin
```

4.3.4. Install binaries from GNU fileutils

```
bash# cd /usr/src/fileutils-4.1
bash# ./configure --host=i386-pc-linux-gnu
bash# make
bash# cd src
bash# cp chgrp chmod chown cp dd df ln ls ~/staging/bin
```

```
bash# cp mkdir mkfifo mknod mv rm rmdir sync ~/staging/bin
```

4.3.5. Install binaries from sh−utils

```
bash# cd /usr/src/sh−utils−2.0
bash# ./configure −−host=i386−pc−linux−gnu
bash# make
bash# cd src
bash# cp date hostname stty su uname ~/staging/bin
```

4.3.6. Copy additional libraries

```
bash# ldd ~/staging/bin/cat
bash# ldd ~/staging/bin/ls
bash# ldd ~/staging/bin/su
bash# ls ~/staging/lib
bash# cp /lib/librt.so.1 ~/staging/lib
bash# cp /lib/libpthread.so.0 ~/staging/lib
bash# cp /lib/libcrypt.so.1 ~/staging/lib
```

4.3.7. Strip binaries and libraries

```
bash# strip ~/staging/bin/*
bash# strip −−strip−unneeded ~/staging/lib/*
```

4.3.8. Create a compressed root disk image

```
bash# cd /
bash# dd if=/dev/zero of=/dev/ram7 bs=1k count=4096
bash# mke2fs −m0 /dev/ram7
bash# mount /dev/ram7 /mnt
bash# cp −dpR ~/staging/* /mnt
bash# umount /dev/ram7
bash# dd if=/dev/ram7 of=~/phase3−image bs=1k
bash# gzip −9 ~/phase3−image
```

4.3.9. Write the root disk image to floppy

Insert the diskette labled "root disk" into drive fd0.

```
bash# dd if=~/phase3−image.gz of=/dev/fd0 bs=1k
```

4.4. Implementation

We will need to have a read−write filesystem in order for some of the commands to work. The kernel's normal behavior is to mount root as read−only, but we can change this using a kernel option. By passing LILO *`rw`* before *`init=/bin/sh`* we will get a read−write root filesystem.

4.4.1. Sytem startup

Follow these steps to get the system running.

- Boot the PC from using the LILO boot disk.
- At the LILO prompt, type **`bootdisk rw init=/bin/sh`** and press **Enter**.
- Insert the recently created root disk when prompted.

The terminal display should look similar to the example below.

```
boot: bootdisk rw init=/bin/sh
Loading bootdisk
Uncompressing Linux... Ok, booting kernel.
..
.. [various kernel messages]
..
VFS: Insert root floppy to be loaded into RAM disk and press ENTER
RAMDISK: Compressed image found at block 0
VFS: Mounted root (ext2 filesystem).
Freeing unused kernel memory: 178k freed
# _
```

4.4.2. Testing new commands

Now that the system is up and running, try using some of the new commands.

```
bash# uname −a
bash# ls /etc
bash# echo "PocketLinux" > /etc/hostname
bash# hostname $(cat /etc/hostname)
bash# uname −n
bash# mkdir /home/stuff
bash# cd /home/stuff
```

If everything goes well the commands like **cat**, **ls** and **hostname** should work now. Even **mkdir** should work since the root filesystem is mounted read−write. Of course since we are using a ramdisk, any changes will be lost once the PC is reset.

4.4.3. System shutdown

Remove the diskette from fd0 and restart the system using **CTRL−ALT−DELETE**.

Chapter 5. Checking and Mounting Disks

5.1. Analysis

In the previous phase of the project we added a lot of new commands and as a result the root disk has a lot more functionality. But there are still a few things lacking. One thing that really stands out is that there was no way to mount disks. In order to get a read−write root filesystem we had to resort to passing the *rw* kernel parameter through LILO. This is fine for an emergency situation, but a normal system boot process should do things differently.

Most GNU/Linux distributions take several steps to mount filesystems. Watching the boot process or digging into the startup scripts on one of the popular Linux distributions reveals the following sequence of events:

1. The kernel automatically mounts the root filesystem as read−only.
2. All local filesystems are checked for errors.
3. If filesystems are clean, root is remounted as read−write.
4. The rest of the local filesystems are mounted.
5. Network filesystems are mounted.

So far our Pocket Linux system can do step one and that is it. If we want to have a professional looking boot / root diskset we will have to do better than one out of five. In this phase of the project we will work on steps two and three. Steps four and five can wait. Since this is a diskette−based system, there really are no other filesystems to mount besides root.

Taking into account all of the above information, the goals for this phase are defined as follows:

- A way to check filesystem integrity.
- The ability to mount filesystems.
- A script to automate checking and mounting of local filesystems.

5.2. Design

5.2.1. Determining necessary utilities.

We can use the Filesystem Hierarchy Standard (FHS) document to help find the names of utilities we need and where they reside in the directory structure. The FHS `/sbin` directory lists **fsck** and something called **fsck.*** for checking filesystems. Since we are using a Second Extended (ext2) filesystem the **fsck.*** becomes **fsck.ext2** for our purposes. Mounting filesystems is done using the commands **mount** and **umount** in the `/bin` directory. However, the name of a script to automatically mount local filesystems cannot be found. On most systems this type of script is in the `/etc` directory, but while FHS does list requirements for `/etc`, it does not currently make recommendations for startup scripts. Several GNU/Linux distributions use `/etc/init.d` as the place to hold startup scripts so we will put our filesystem mounting script there.

5.2.2. Finding source code

If we search Ibiblio's Linux Software Map (LSM) at http://www.ibiblio.org/pub/Linux/ for the keyword "fsck" we get a large number of matches. Since we are using a Second Extended filesystem, called ext2 for short, we can refine the search using "ext2" as a keyword. Supplying both keywords to the LSM search engine comes

up with a package called e2fsprogs. Looking at the LSM entry for e2fsprogs we find out that package contains the utilities **e2fsck**, **mke2fs**, **dumpe2fs**, **fsck** and more. We also find out that the LSM entry for e2fsprogs has not been updated since 1999. There is almost certainly a newer version out there somewhere. Another good Internet resource for source code is SourceForge at http://sourceforge.net. Using the keyword "e2fsprogs" in the SourceForge search engine results in a much newer version of e2fsprogs.

Finding **fsck** was quite an adventure, but now we can move on to finding **mount** and **umount**. A search on LSM comes up with a number of matches, but most of them point to various versions of a package called util−linux. All we have to do is scroll through and pick the most recent release. The LSM entry for util−linux lists a lot of utilities besides just mount and umount. We should definitely scan through the list to see if any of the other util−linux commands show up in the FHS requirements for `/bin` and `/sbin`.

Below is a list of packages we have gathered so far and the utilities that match up with FHS.

- e2fsprogs −− **fsck**, **fsck.ext2** (**e2fsck**), **mkfs.ext2** (**mke2fs**)
- util−linux −− **dmesg**, **getty** (**agetty**), **kill**, **login**, **mount**, **swapon**, **umount**

5.2.3. Automating fsck and mount

Now that we have **fsck** and **mount** commands we need to come up with a shell script to automate checking and mounting the local filesystems. An easy way to do this would be to write a short, two line script that calls **fsck** and then **mount**. But, what if the filesystems are not clean? The system should definitely not try to mount a corrupted filesystem. Therefore we need to devise a way of determining the status of the filesystems before mounting them. The manpage for **fsck** gives some insight into how this can be accomplished using return codes. Basically, a return code of zero or one means the filesystem is okay and two or greater means some kind of manual intervention is needed. A simple if−then statement could evaluate the **fsck** return code to determine whether or not the filesystem should be mounted. For help on writing shell scripts we can turn to the BASH(1) manpage and the Advanced BASH Scripting Guide. Both references are freely available from the Linux Documentation Project at http://www.tldp.org.

5.2.4. File dependencies

The last thing to do is to figure out if any other files besides the binaries are needed. We learned about using **ldd** to check for library dependencies in the last phase of the project and we will use it to check the utilities in this phase too. There are also some other files that **fsck** and **mount** will need and the fsck(8) and mount(8) manpages give some insight into what those files are. There is `/etc/fstab` that lists devices and their mount points, `/etc/mtab` which keeps track of what is mounted and the device files that represent the various disks. We will need to include all of these to have everything work right.

The `/etc/fstab` file is just a simple text file that can be created with any editor. We will need an entry for the root filesystem and for the proc filesystem. The reason for the proc filesystem entry is so we can create `/etc/mtab` as a symlink that points to `/proc/mounts`. The `/proc/mounts` file contains almost exactly the same information as the traditional `/etc/mtab` file. We just have to make sure the proc filesystem is mounted before anything else. The only thing left is to create device files. We will need `/dev/ram0`, because that is where the root filesystem is located. We also need `/dev/fd0` to mount other floppy disks and `/dev/null`.

5.3. Construction

5.3.1. Install utilities from e2fsprogs

```
bash# cd /usr/src/e2fsprogs-1.29
bash# ./configure --host=i386-pc-linux-gnu
bash# make
bash# cd e2fsck
bash# cp e2fsck.shared ~/staging/sbin/e2fsck
bash# ln -s e2fsck ~/staging/sbin/fsck.ext2
bash# cd ../misc
bash# cp fsck mke2fs ~/staging/sbin
bash# ln -s mke2fs ~/staging/sbin/mkfs.ext2
```

5.3.2. Install utilities from util-linux

```
bash# cd /usr/src/util-linux-2.11u
```

Use a text editor to make the following changes to `MCONFIG`:

- Change "CPU=$(shell uname -m)" to "CPU=i386"
- Change "HAVE_SHADOW=yes" to "HAVE_SHADOW=no"

```
bash# ./configure
bash# make
bash# cp disk-utils/mkfs ~/staging/sbin
bash# cp fdisk/fdisk ~/staging/sbin
bash# cp login-utils/agetty ~/staging/sbin
bash# ln -s agetty ~/staging/sbin/getty
bash# cp login-utils/login ~/staging/bin
bash# cp misc-utils/kill ~/staging/bin
bash# cp mount/mount ~/staging/bin
bash# cp mount/umount ~/staging/bin
bash# cp mount/swapon ~/staging/sbin
bash# cp sys-utils/dmesg ~/staging/bin
```

5.3.3. Check library requirements

```
bash# ldd ~/staging/bin/* | more
bash# ldd ~/staging/sbin/* | more
bash# ls ~/staging/lib
```

All of the dependencies revealed by the **ldd** command are for libraries already present in the staging area so there is no need to copy anything new.

5.3.4. Strip binaries to save space

```
bash# strip ~/staging/bin/*
bash# strip ~/staging/sbin/*
```

5.3.5. Create additional device files

```
bash# mknod ~/staging/dev/ram0 b 1 0
bash# mknod ~/staging/dev/fd0 b 2 0
bash# mknod ~/staging/dev/null c 1 3
```

5.3.6. Create mtab and fstab files

```
bash# cd ~/staging/etc
bash# ln −s /proc/mounts mtab
```

Use an editor like vi emacs or pico to create the following file and save it as `~/staging/etc/fstab`.

```
proc            /proc   proc    noauto          0       0
/dev/ram0       /       ext2    defaults        1       1
```

5.3.7. Write a script to mount the proc filesystem

```
bash# mkdir ~/staging/etc/init.d
bash# cd ~/staging/etc/init.d
```

Use an editor to create the following shell script and save it as `~/staging/etc/init.d/proc_fs`:

```
#!/bin/sh
#
# proc_fs − mount the proc filesystem
#
PATH=/sbin:/bin ; export PATH

mount −t proc proc /proc
#
# end of proc_fs
```

5.3.8. Write a script to check and mount local filesystems

Use an editor to create the following shell script and save it as `~/staging/etc/init.d/local_fs`:

```
#!/bin/sh
#
# local_fs − check and mount local filesystems
#
PATH=/sbin:/bin ; export PATH

fsck −ATCp
if [ $(($?)) −gt $((1)) ]; then
  echo "Filesystem errors still exist!  Manual intervention required."
  /bin/sh
else
  echo "Remounting / as read−write."
  mount −o remount,rw /
  echo "Mounting local filesystems."
  mount −a
fi
#
# end of local_fs
```

Set execute permissions on the scripts.

```
bash# chmod +x proc_fs
bash# chmod +x local_fs
```

5.3.9. Create a compressed root disk image

```
bash# cd /
bash# dd if=/dev/zero of=/dev/ram7 bs=1k count=4096
bash# mke2fs −m0 /dev/ram7
bash# mount /dev/ram7 /mnt
bash# cp −dpR ~/staging/* /mnt
bash# umount /dev/ram7
bash# dd if=/dev/ram7 of=~/phase4−image bs=1k
bash# gzip −9 ~/phase4−image
```

5.3.10. Write the root disk image to floppy

Insert the diskette labled "root disk" into drive fd0.

```
bash# dd if=~/phase4−image.gz of=/dev/fd0 bs=1k
```

5.4. Implementation

5.4.1. System startup

Start the system using the following procedure:

- Boot the PC using the floppy labled "boot disk".
- Type **`bootdisk init=/bin/sh`** at the LILO prompt.
- Put in the recently created root disk when prompted.

The output should resemble the example below:

```
boot: bootdisk init=/bin/sh
Loading bootdisk
Uncompressing Linux... Ok, booting kernel.
..
.. [various kernel messages]
..
VFS: Insert root floppy to be loaded into RAM disk and press ENTER
RAMDISK: Compressed image found at block 0
VFS: Mounted root (ext2 filesystem) readonly.
Freeing unused kernel memory: 178k freed
# _
```

5.4.2. Test proc_fs and local_fs scripts

Run the scripts by typing the following commands at the shell prompt:

```
bash# PATH=/sbin:/bin:/etc/init.d ; export PATH
bash# proc_fs
```

```
bash# cat /etc/mtab
bash# local_fs
bash# df
```

If everything is working properly, then the screen output should look something like the example below.

```
bash# PATH=/sbin:/bin:/etc/init.d ; export PATH
bash# proc_fs
bash# cat /etc/mtab
/dev/root / ext2 ro 0 0
proc /proc proc rw 0 0
bash# local_fs
/dev/ram0: clean 74/1024 files 3178/4096 blocks
Remounting / as read-write.
Mounting local filesystems.
bash# df
Filesystem      1k-blocks       Used Available Use% Mounted on
/dev/root       3963            3045 918        77% /
```

5.4.3. Create and mount additional filesystems

Remove the root disk floppy and insert a blank diskette labeled "home". Then type the following commands:

```
bash# mkfs /dev/fd0
bash# fsck /dev/fd0
bash# mount /dev/fd0 /home
bash# mkdir /home/floyd
bash# cd /home/floyd
bash# echo "Goodbye cruel world." > goodbye.txt
bash# cat goodbye.txt
```

5.4.4. System shutdown

```
bash# cd /
bash# umount /dev/fd0
bash# sync
```

Remove the diskette from fd0 and restart the system using **CTRL-ALT-DELETE**.

Chapter 6. Automating Startup & Shutdown

6.1. Analysis

The root disk from the last chapter is looking pretty good. It has about seventy percent of the commands that the Filesystem Hierarchy Standard (FHS) document requires for the root filesystem. Plus it has commands for checking and mounting filesystems. But even with all of this the root disk is far from perfect. The list below outlines three things that could use some improvement if the Pocket Linux system is to stand up next to the more professional looking distributions.

1. The system currently requires the kernel parameter *`init=/bin/sh`* to be typed at the LILO prompt in order to start properly. On any other GNU/Linux system this is only done in an emergency situation when the system is corrupted.
2. Checking and mounting the root filesystem has to be done manually by running a script at a shell prompt. On most modern operating systems this function is handled automatically as part of the system start−up process.
3. Using **CTRL−ALT−DELETE** for system shutdown is not very graceful. Filesystems should be unmounted and cached information should be flushed prior to shutdown. Again, this is something that most operating systems handle automatically.

Taking the above list into consideration, the goals for this phase are defined as follows:

- Automated start−up sequence.
- Graceful shutdown capability.

6.2. Design

6.2.1. Determining necessary utilities

We need an init daemon to automate start−up. We know this because the Bootdisk−HOWTO and From−Powerup−To−BASH−Prompt−HOWTO both make mention of it as the first program to start after the kernel loads. The latter HOWTO also goes into some detail about the `/etc/inittab` file and the organization of startup scripts. This could be helpful since FHS, the blueprint we have used so far, makes no recommendation for init scripts.

We will also need to find the shutdown command to fulfill the second goal of graceful shutdown capability.

6.2.2. Obtaining source code

Searching the Linux Software Map on Ibiblio for the keyword "init" gives a large number of results. From reading the From−Powerup−To−BASH−Prompt−HOWTO however, we know that most Linux systems use a System V style init daemon. Narrowing the search with the additional key phrase of "System V" gives much better results. The sysvinit package contains **init**, **shutdown**, **halt** and **reboot** which is everything we need. The version listed in the LSM entry looks to be pretty old, but there is a primary−site URL that will probably lead to the latest version.

6.2.3. Checking Dependencies

The manpage for init mentions a FIFO called `/dev/initctl` that is required for **init** to communicate with other programs in the sysvinit package. We will have to create this file for **init** to function properly.

6.2.4. Outlining start−up scripts

Many of the popular GNU/Linux distributions use System V style init scripts. Since we are using a "sysvinit" daemon it makes since to use System V style scripts as well. The following documents all touch upon the System V style init scripts in some way and will serve as references when building the scripts for this project:

- The Debian Policy Manual −− available online at http://www.debian.org/doc/debian−policy.
- The Linux Standard Base specification −− downloadable in many formats from http://www.linuxbase.org/spec/index.shtml.
- Essential System Administration, 3rd Edition by Aeleen Frisch −− available at bookstores or directly from O'Reilly Publishing at http://www.oreilly.com/.

After glancing at one or two of the above references we should have a pretty good idea of how the System V style system initialization process works. We should also know what it takes to create System V style init scripts for the Pocket Linux project. Below is a brief list of what needs to be done:

- Create an `inittab` file to call an `rc` script with a numerical argument giving the runlevel.
- Write an `rc` script that use the runlevel argument to execute the appropriate "K" and "S" scripts.
- Modify the previously built `local_fs` script to take *`start`* and *`stop`* arguments.
- Create new scripts for `shutdown` and `reboot`.
- Set up `/etc/rcN.d` directories and links to scripts in `/etc/init.d`.

As always, the BASH(1) manpage and the Advanced BASH Scripting Guide are very helpful for writing and understanding shell scripts.

6.3. Construction

6.3.1. Install sysvinit utilities

```
bash# cd /usr/src/sysvinit−2.84/src
bash# make clobber
bash# make CC="gcc −mcpu=i386"
bash# cp halt init shutdown ~/staging/sbin
bash# ln −s halt ~/staging/sbin/reboot
bash# ln −s init ~/staging/sbin/telinit
bash# mknod ~/staging/dev/initctl p
```

In the interest of speed we are skipping the steps for checking libraries and stripping binaries. The library requirements for sysvinit are very basic and the Makefile is configured to automatically strip the binaries.

6.3.2. Create /etc/inittab file

Use a text editor to create the following file and save it as `~/staging/etc/inittab`

```
# /etc/inittab − init daemon configuration file
#
# Default runlevel
id:1:initdefault:
#
# System initialization
si:S:sysinit:/etc/init.d/rc S
#
# Runlevel scripts
r0:0:wait:/etc/init.d/rc 0
r1:1:respawn:/bin/sh
r2:2:wait:/etc/init.d/rc 2
r3:3:wait:/etc/init.d/rc 3
r4:4:wait:/etc/init.d/rc 4
r5:5:wait:/etc/init.d/rc 5
r6:6:wait:/etc/init.d/rc 6
#
# end of /etc/inittab
```

6.3.3. Create /etc/init.d/rc script

Use a text editor to create the following file and save it as `~/staging/etc/init.d/rc`

```
#!/bin/sh
#
# /etc/init.d/rc − runlevel change script
#
PATH=/sbin:/bin
SCRIPT_DIR="/etc/rc$1.d"
#
# Check that the rcN.d directory really exists.
if [ −d $SCRIPT_DIR ]; then
#
# Execute the kill scripts first.
  for SCRIPT in $SCRIPT_DIR/K*; do
    if [ −x "$SCRIPT" ]; then
      $SCRIPT stop;
    fi;
  done;
#
# Do the Start scripts last.
  for SCRIPT in $SCRIPT_DIR/S*; do
    if [ −x $SCRIPT ]; then
      $SCRIPT start;
    fi;
  done;
fi
```

Make the file executable.

```
bash# chmod +x ~/staging/etc/init.d/rc
```

6.3.4. Modify /etc/init.d/local_fs script

A case statement is added to allow the script to either mount or dismount local filesystems depending on the command−line argument given. The original script is contained inside the "start" portion of the case statement. The "stop" portion is new.

```
#!/bin/sh
#
# local_fs − check and mount local filesystems
#
PATH=/sbin:/bin ; export PATH

case $1 in

start)
  echo "Checking local filesystem integrity."
  fsck −ATCp
  if [ $(($?)) −gt $((1)) ]; then
    echo "Filesystem errors still exist!  Manual intervention required."
    /bin/sh
  else
    echo "Remounting / as read−write."
    mount −o remount,rw /
    echo "Mounting local filesystems."
    mount −a
  fi
;;

stop)
  echo "Dismounting local filesystems."
  umount −a
  echo "Remounting / as read−only."
  mount −o remount,rw /
  echo "Flushing disk cache."
  sync
;;

default)
  echo "usage: $0 start|stop";
;;

esac
#
# end of local_fs
```

6.3.5. Create a hostname script

Use a text editor to create the following script and save it as `~/staging/etc/init.d/hostname`

```
#!/bin/sh
#
# hostname − set the system name to the name stored in /etc/hostname
#
PATH=/sbin:/bin ; export PATH

echo "Setting hostname."
if [ −f /etc/hostname ]; then
  hostname $(cat /etc/hostname)
```

```
else
  hostname gnu-linux
fi
#
# end of hostname
```

6.3.6. Create halt & reboot scripts

Use a text editor to create `~/staging/etc/init.d/halt` as shown below.

```
#!/bin/sh
#
# halt - halt the system
#
PATH=/sbin:/bin ; export PATH

echo "Initiating system halt."
halt
#
# end of /etc/init.d/halt
```

Create the following script and save it as `~/staging/etc/init.d/reboot`

```
#!/bin/sh
#
# reboot - reboot the system
#
PATH=/sbin:/bin ; export PATH

echo "Initiating system reboot."
reboot
#
# end of /etc/init.d/reboot
```

Flag script files as executable.

```
bash# chmod +x ~/staging/etc/init.d/*
```

6.3.7. Create rcN.d directories and links

```
bash# cd ~/staging/etc
bash# mkdir rc0.d rc1.d rc2.d rc3.d rc4.d rc5.d rc6.d rcS.d
bash# cd ~/staging/etc/rcS.d
bash# ln -s ../init.d/proc_fs S10proc_fs
bash# ln -s ../init.d/local_fs S20local_fs
bash# ln -s ../init.d/hostname S30hostname
bash# cd ~/staging/etc/rc0.d
bash# ln -s ../init.d/local_fs K10local_fs
bash# ln -s ../init.d/halt K90halt
bash# cd ~/staging/etc/rc6.d
bash# ln -s ../init.d/local_fs K10local_fs
bash# ln -s ../init.d/reboot K90reboot
```

6.3.8. Create the root disk image

```
bash# cd /
bash# dd if=/dev/zero of=/dev/ram7 bs=1k count=4096
bash# mke2fs −m0 /dev/ram7
bash# mount /dev/ram7 /mnt
bash# cp −dpR ~/staging/* /mnt
bash# umount /dev/ram7
bash# dd if=/dev/ram7 of=~/phase5−image bs=1k
bash# gzip −9 ~/phase5−image
```

6.3.9. Copy the image to diskette

Insert the diskette labled "root disk" into drive fd0.

```
bash# dd if=~/phase5−image.gz of=/dev/fd0 bs=1k
```

6.4. Implementation

6.4.1. System Startup

Boot the PC using the floppy labled "boot disk" and press **Enter** at the LILO prompt. Place the recently created root disk in fd0 when prompted. The output should resemble the example below:

```
boot: bootdisk
Loading bootdisk
Uncompressing Linux... Ok, booting kernel.
..
.. [various kernel messages]
..
VFS: Insert root floppy to be loaded into RAM disk and press ENTER
RAMDISK: Compressed image found at block 0
VFS: Mounted root (ext2 filesystem) readonly.
Freeing unused kernel memory: 178k freed
Checking local filesystem integrity.
/dev/ram0: clean 105/1024 files 2842/4096 blocks
Remounting / as read−write.
Mounting local filesystems.
Setting the hostname.
INIT: Entering runlevel: 1
# _
```

6.4.2. Verify success of startup scripts

Use the **mount** command to check that local filesystems are mounted as read−write. The output should look like the example below.

```
bash# mount
/dev/root on / type ext2 (rw)
proc on /proc type proc (rw)
```

Check the hostname.

```
bash# uname -n
gnu-linux
```

6.4.3. System shutdown

Bring the system down gracefully with the **shutdown** command.

```
bash# shutdown -h now
```

We should see the following output from **init** and the shutdown scripts:

```
INIT: Switching to runlevel: 0
INIT: Sending processes the TERM signal
Terminated
INIT: Sending processes the KILL signal
Dismounting local filesystems.
Remounting / as read-only.
Flushing disk cache.
Initiating system halt.
System halted.
```

Chapter 7. Enabling Multiple Users

7.1. Analysis

Up to now the system has been operating in single−user mode. There is no login process and anyone who boots the system goes straight into a shell with root privileges. Obviously, this is not the normal operating mode for most GNU/Linux distributions. Most systems feature multi−user capability where many users can access the system simultaneously with different privilege levels. These multi−user systems also support virtual consoles so that the keyboard and video display can be multiplexed between several terminal sessions. So in this phase we would like to add the following enhancements to the system:

- Enable multi−user capability.
- Create multiple, virtual consoles.

7.2. Design

7.2.1. The login process

The From−Powerup−To−BASH−Prompt−HOWTO does a good job of outlining the steps in the login process. Basically it works like this.

1. The **init** daemon starts a **getty** process on the terminal.
2. The **getty** program displays the contents of `/etc/issue` and prompts for a user name.
3. When the user name is entered, control is handed off to the **login** program.
4. The **login** program asks for a password and verifies the credentials using `/etc/passwd`, `/etc/group` and possibly `/etc/shadow`.
5. If everything is okay the user's shell is started.

7.2.2. Obtaining source code

The **getty** and **login** programs were already installed as part of util−linux.

7.2.3. Creating support files

7.2.3.1. Device nodes

Details about virtual console device files can be found in the Linux kernel source code file called `devices.txt` in the `Documentation` directory. We will need to create `tty1` through `tty6` for each of the virtual consoles as well as `tty0` and `tty` to represent the current virtual console.

7.2.3.2. /etc/issue

The `/etc/issue` file is pretty easy to construct. It can contain any text we want displayed on the screen prior to the login prompt. It could be something friendly like "Welcome to Pocket Linux", something menacing like "Authorized users only!" or it something informational like "Connected to tty1 at 9600bps". The agetty(8) manpage explains how to display information like tty line and baud rate using escape codes.

7.2.3.3. /etc/passwd

The format of `/etc/passwd` can be obtained by reading the passwd(5) manpage. We can easily create a user account by adding a line like "root::0:0:superuser:/root:/bin/sh" to the file.

Maintaining passwords will be somewhat challenging because of the system being loaded into ramdisk. Any changes to `/etc/passwd` will be lost when the system is shutdown. So to make things easy, we will create all users with null passwords.

7.2.3.4. /etc/group

The structure of `/etc/group` is available from the group(5) manpage. A line of "root::0:root" would define a group called "root" with no password, a group id of zero and the user root assigned to it as the only member.

7.2.3.5. Conventions

User and group names and id's are generally not chosen at random. Most Linux systems have very similar looking `/etc/passwd` and `/etc/group` files. Definitions for commonly used user id and group id assignments may be found in one of several places:

- The `/etc/passwd` and `/etc/group` files on any popular GNU/Linux distribution.
- The Debian Policy Manual −− available online at http://www.debian.org/doc/debian−policy.
- The Linux Standard Base specification −− downloadable in many formats from http://www.linuxbase.org/spec/index.shtml.
- Essential System Administration, 3rd Edition by Aeleen Frisch −− available at bookstores or directly from O'Reilly Publishing at http://www.oreilly.com/.

7.2.4. Dependencies

Running **ldd** on the `login` program from util−linux will reveal that it is linked to the library `libcrypt.so.1`. In addition to libcrypt, there is another, less obvious library dependency on `libnss_files.so.2`. The name service switch library `libnss_files.so.2` is required for the login program to access the `/etc/passwd` file. Without libnss, all logins will mysteriously fail.

7.2.5. Assigning ownership and permissions

Previously, with the single user system, there was no need to worry about permissions when installing directories, files and device nodes. The shell was effectively operating as root, so everything was accessible. Things become more complex with the addition of multiple user capability. Now we need to make sure that every user has access to what they need and at the same time gets blocked from what they do not need.

A good guideline for assigning ownership and permissions would be to give the minimum level of access required. Take the `/bin` directory as an example. The Filesystem Hierarchy (FHS) document says, "`/bin` contains commands that may be used by both the system administrator and by users". From that statement we can infer that `/bin` should have read and execute permission for everyone. On the other hand, the `/boot` directory contains files for the boot loader. Chances are good that regular users will not need to access anything in the `/boot` directory. So the minimum level of access would be read permission for the root user and other administrators who are members of the root group. Normal users would have no permissions assigned on the `/boot` directory.

Most of the time we can assign similar permissions to all the commands in a directory, but there are some programs that prove to be exceptions to the rule. The **su** command is a good example. Other commands in the /bin directory have a minimum requirement of read and execute, but the **su** command needs to be setuid root in order to run correctly. Since it is a setuid binary, it might be a good idea not to allow just anyone to run it. Ownership of 0:0 (root user, root group) and permissions of rwsr−x−−− (octal 1750) would be a good fit for **su**.

The same logic can be applied to other directories and files in the root filesystem using the following steps:

1. Assign ownership to the root user and root group.
2. Set the most restrictive permissions possible.
3. Adjust ownership and permissions on an "as needed" basis.

7.3. Construction

7.3.1. Verify presence of getty and login

```
bash# ls ~/staging/sbin/getty
bash# ls ~/staging/bin/login
```

7.3.2. Modify inittab for multi−user mode

Modify `~/staging/etc/inittab` by changing the default runlevel and adding **getty** entries as shown below.

```
# /etc/inittab − init daemon configuration file
#
# Default runlevel
id:2:initdefault:
#
# System initialization
si:S:sysinit:/etc/init.d/rc S
#
# Runlevel scripts
r0:0:wait:/etc/init.d/rc 0
r1:1:respawn:/bin/sh
r2:2:wait:/etc/init.d/rc 2
r3:3:wait:/etc/init.d/rc 3
r4:4:wait:/etc/init.d/rc 4
r5:5:wait:/etc/init.d/rc 5
r6:6:wait:/etc/init.d/rc 6
#
# Spawn virtual terminals
1:235:respawn:/sbin/getty 9600 tty1 linux
2:235:respawn:/sbin/getty 9600 tty2 linux
3:235:respawn:/sbin/getty 9600 tty3 linux
4:235:respawn:/sbin/getty 9600 tty4 linux
5:235:respawn:/sbin/getty 9600 tty5 linux
6:2345:respawn:/sbin/getty 9600 tty6 linux
#
# end of /etc/inittab
```

7.3.3. Create tty devices

```
bash# cd ~/staging/dev
bash# mknod tty0 c 4 0
bash# mknod tty1 c 4 1
bash# mknod tty2 c 4 2
bash# mknod tty3 c 4 3
bash# mknod tty4 c 4 4
bash# mknod tty5 c 4 5
bash# mknod tty6 c 4 6
bash# mknod tty c 5 0
```

7.3.4. Create support files in /etc

7.3.4.1. /etc/issue

Create the file `~/staging/etc/issue` using the example below or design a customized message.

```
Connected to \l at \b bps.
```

Note that "\l" is a lowercase letter L, not the number one.

7.3.4.2. /etc/passwd

Use a text editor to create a minimal passwd file conforming to the Linux Standards Base (LSB) document. Save the file as `~/staging/etc/passwd`

```
root::0:0:Super User:/root:/bin/sh
bin:x:1:1:Legacy UID:/bin:/bin/false
daemon:x:2:2:Legacy UID:/sbin:/bin/false
```

7.3.4.3. /etc/group

Use a text editor to create an LSB conforming group file and save it as `~/staging/etc/group`

```
root::0:root
bin:x:1:root,bin,daemon
daemon:x:2:root,bin,daemon
```

7.3.5. Copy required libraries

```
bash# cp /lib/libnss_files.so.2 ~/staging/lib
```

7.3.6. Set directory and file permissions

Set minimal privileges on all files and directories under `~/staging`. Everything is owned by the root user and the root group. Permissions are read−write for the owner and read−only for the group. Exceptions to the blanket permissions are handled case by case.

```
bash# cd ~/staging
bash# chown −R 0:0 *
```

```
bash# chmod −R 640 *
```

Set execute permission on all directories. (Note the capital "X")

```
bash# chmod −R +X *
```

Files in `/bin` are read and execute for all, but `su` is an exception.

```
bash# chmod 755 bin/*
bash# chmod 4750 bin/su
```

Files in `/dev` have various permissions. Disk devices should be accessible to administrators only. Other files like `/dev/null` should have full privileges granted to everyone.

```
bash# chmod 660 dev/fd0 dev/ram0
bash# chmod 666 dev/null
bash# chmod 622 dev/console
bash# chmod 600 dev/initctl
bash# chmod 622 dev/tty
bash# chmod 622 dev/tty?
```

The `passwd` and `group` files must be world readable.

```
bash# chmod 644 etc/passwd
bash# chmod 644 etc/group
```

The scripts in `/etc/init.d` are read and execute for administrators.

```
bash# chmod 750 etc/init.d/*
```

Libraries need read and execute permissions for everyone.

```
bash# chmod 755 lib/*
```

Only root should have access to the `/root` directory.

```
bash# chmod 700 root
```

Make files in `/sbin` read and execute for administrators.

```
bash# chmod 750 sbin/*
```

Temp should be read−write for all with the sticky bit set.

```
bash# chmod 1777 tmp
```

7.3.7. Create the root disk image

```
bash# cd /
bash# dd if=/dev/zero of=/dev/ram7 bs=1k count=4096
bash# mke2fs −m0 /dev/ram7
bash# mount /dev/ram7 /mnt
```

```
bash# cp −dpR ~/staging/* /mnt
bash# umount /dev/ram7
bash# dd if=/dev/ram7 of=~/phase6−image bs=1k
bash# gzip −9 ~/phase6−image
```

7.3.8. Copy the image to diskette

Insert the diskette labled "root disk" into drive fd0.

```
bash# dd if=~/phase6−image.gz of=/dev/fd0 bs=1k
```

7.4. Implementation

7.4.1. System Startup

If everything goes well, the virtual console display should look similar to the following example:

```
Connected to tty1 at 9600 bps.
gnu−linux login:
```

7.4.2. Add a new user to the system

Log in as root.

Create a new, unprivileged user and new group by appending a line to the `/etc/passwd` and `/etc/group` files, respectively. Be sure to use a double greater−than (>>) to avoid accidentally overwriting the files.

```
bash# echo "floyd::501:500:User:/home/floyd:/bin/sh" >> /etc/passwd
bash# echo "users::500:" >> /etc/group
bash# mkdir /home/floyd
bash# chown floyd.users /home/floyd
bash# chmod 700 /home/floyd
```

7.4.3. Test the new user's ability to use the system

Switch to virtual terminal tty2 by pressing **ALT**+**F2**.

Log in as floyd.

Try the following commands and verify that they work.

```
bash$ pwd
bash$ ls −l /
bash$ cat /etc/passwd
```

Try the following commands and verify that they do not work.

```
bash$ ls /root
bash$ /sbin/shutdown −h now
bash$ su −
```

7.4.4. System shutdown

Switch back to tty1 where root is logged in.

```
bash# shutdown -h now
```

Chapter 8. Filling in the Gaps

8.1. Analysis

The root disk has come a long way since its humble beginnings as a statically−linked shell. It now shares many features with the popular, ready−made distributions. For example it has:

- Several common utilities like **cat**, **ls** and so on.
- Startup scripts that automatically check and mount filesystems.
- Graceful shutdown capability.
- Support for multiple users and virtual terminals.

As a final test, we can put the root disk up against the Filesystem Hierarchy Standard (FHS) requirements for the root filesystem. (We will ignore anything in the `/usr` hierarchy because of space constraints.) Compared to FHS requirement, the only files missing are a few commands in the `/bin` directory. Specifically, the root disk lacks the following commands:

- **more**
- **ps**
- **sed**

In addition to the required commands, it might be nice to include the "ed" editor listed as an option by the FHS. It is not as robust as vi or emacs, but it works and it should fit onto the tiny root filesystem.

So in order to finish up this phase of the project, we need to accomplish the following goals:

- Add the **more**, **ps** and **sed** commands.
- Install the optional **ed** editor.

8.2. Design

8.2.1. more

There is a **more** command that comes with util−linux, but it will not work for this project. The reason is because of library dependencies and space constraints. The util−linux supplied **more** needs either the libncurses or libtermcap to work and there just is not enough space on the root disk floppy to fit everything in. So, in order to have a **more** command we will have to get creative.

The **more** command is used to display a file page−by−page. It's a little like having a **cat** command that pauses every twenty−five lines. The basic logic is outlined below.

- Read one line of the file.
- Display the line on the screen.
- If 25 lines have been displayed, pause.
- Loop and do it again.

Of course there are some details left out like what to do if the screen dimensions are not what we anticipated, but overall it is a fair representation of what **more** does. Given this simple program logic, it should not be hard

to put together a short shell script that emulates the basic functionality of **more**. The BASH(1) manpage and Adv−BASH−Scripting−Guide will serve as references.

8.2.2. More device files

The **more** script will need access to device files that are not on the root disk yet. Specifically **more** needs to have `stdin`, `stdout` and `stderr`, but while we are at it we should check for any other missing `/dev` files. The Linux Standard Base requires `null`, `zero` and `tty` to be present in the `/dev` directory. Files for `null` and `tty` already exist from previous phases of the project, but we still need `/dev/zero`. We can refer to `devices.txt` in the Linux source code `Documentation` directory for major and minor numbers.

8.2.3. ps, sed & ed

These three packages can be found by using some of the same Internet resources we have used before. The procps package shows up in an Ibiblio LSM search (http://www.ibiblio.org/pub/linux/) for the keyword "ps". The "sed" and "ed" packages can be found on the GNU ftp server at ftp://ftp.gnu.org.

Both "sed" and "ed" packages feature GNU's familiar **configure** script and are therefore very easy to build. There is no **configure** script for "procps" but this does not make things too difficult. We can just read the package's `INSTALL` file to find out about how to set various configuration options. We can use one of these options to avoid the complexity of using and installing libproc. Setting *`SHARED=0`* makes `libproc` an integrated part of **ps** rather than a separate, shared library.

8.3. Construction

8.3.1. Write a "more" script

Create the following script with a text editor and save it as `~/staging/bin/more.sh`

```
#!/bin/sh
#
# more.sh − emulates the basic functions of the "more" binary without
#           requiring ncurses or termcap libraries.
#
# Assume input is coming from STDIN unless a valid file is given as
# a command−line argument.
if [ −f "$1" ]; then
  INPUT="$1"
else
  INPUT="/dev/stdin"
fi
#
# Set IFS to newline only. See BASH(1) manpage for details on IFS.
IFS=$'\n'
#
# If terminal dimensions are not already set as shell variables, take
# a guess of 80x25.
if [ "$COLS" = "" ]; then
  COLS=80;
fi
if [ "$ROWS" = "" ]; then
  ROWS=25;
fi
```

```
#
# Initialize row counter variable
ROW_COUNTER=$ROWS
#
# Read the input file one line at a time and display on STDOUT until
# the page fills up. Display "Press <Enter>" message on STDERR and wait
# for keypress from STDERR.  Continue until the end of the input file.
# Any input line greater than $COLS characters in length is wrapped and
# counts as multiple lines.
#
while read −n $COLS LINE_BUFFER; do
  echo "$LINE_BUFFER"
  ROW_COUNTER=$(($ROW_COUNTER − 1))
  if [ $ROW_COUNTER −le 1 ]; then
    echo "Press <ENTER> for next page or <CTRL>+C to quit.">/dev/stderr
    read</dev/stderr
    ROW_COUNTER=$ROWS
  fi
done<$INPUT
#
# end of more.sh
```

Create a symbolic link for `more`

```
bash# ln −s more.sh ~/staging/bin/more
```

8.3.2. Create additional device files

```
bash# ln −s /proc/self/fd ~/staging/dev/fd
bash# ln −s fd/0 ~/staging/dev/stdin
bash# ln −s fd/1 ~/staging/dev/stdout
bash# ln −s fd/2 ~/staging/dev/stderr
bash# mknod −m644 ~/staging/dev/zero c 1 5
```

8.3.3. Install procps

```
bash# cd /usr/src/procps−2.07
bash# make SHARED=0 CC="gcc −mcpu=i386"
bash# cd ps
bash# cp ps ~/staging/bin
```

8.3.4. Install sed

```
bash# cd /usr/src/sed−3.02
bash# ./configure −−host=i386−pc−linux−gnu
bash# make
bash# cd sed
bash# cp sed ~/staging/bin
```

8.3.5. Install ed

```
bash# cd /usr/src/ed−0.2
bash# ./configure −−host=i386−pc−linux−gnu
bash# make
bash# cp ed ~/staging/bin
```

8.3.6. Strip binaries to save space

```
bash# strip ~/staging/bin/*
```

8.3.7. Ensure proper permissions

```
bash# chown 0:0 ~/staging/bin/*
bash# chmod −R 755 ~/staging/bin
bash# chmod 4750 ~/staging/bin/su
```

8.3.8. Create the root disk image

```
bash# cd /
bash# dd if=/dev/zero of=/dev/ram7 bs=1k count=4096
bash# mke2fs −m0 /dev/ram7
bash# mount /dev/ram7 /mnt
bash# cp −dpR ~/staging/* /mnt
bash# umount /dev/ram7
bash# dd if=/dev/ram7 of=~/phase7−image bs=1k
bash# gzip −9 ~/phase7−image
```

8.3.9. Copy the image to diskette

Insert the diskette labled "root disk" into drive fd0.

```
bash# dd if=~/phase7−image.gz of=/dev/fd0 bs=1k
```

8.4. Implementation

8.4.1. System startup

Boot from the diskset in the usual way and log in as root.

8.4.2. Test the "more" script

Display kernel messages by piping the output of **dmesg** to **more**.

```
bash# dmesg | more
```

Examine the `local_fs` script by using **more** with a command−line argument.

```
bash# more /etc/init.d/local_fs
```

8.4.3. Use ps to show running processes

Display processes for the user currently logged in.

```
bash# ps
```

Display all available information about all running processes.

```
bash# ps −ef
```

8.4.4. Run a simple sed script

Use **sed** to display an alternate version of `/etc/passwd`.

```
bash# sed −e "s/Legacy/Old School/" /etc/passwd
```

Verify that sed did not make the changes permanent.

```
bash# cat /etc/passwd
```

8.4.5. Test the "ed" editor

Use **ed** to change properties on the "daemon" user.

```
bash# ed −p*
ed* r /etc/passwd
ed* %p
ed* /daemon/s/Legacy/Old School/
ed* %p
ed* w
ed* q
```

Verify that the changes are permanent (at least until the system is restarted.)

```
bash# cat /etc/passwd
```

8.4.6. System shutdown

Bring the system down gracefully with the **shutdown** command.

Chapter 9. Project Wrap Up

9.1. Celebrating Accomplishments

As the Pocket Linux Project draws to a close we should take a moment to celebrate all of our accomplishments. Some of the highlights are listed below:

- We have built a system, from source code only, that fully implements all of the commands described in the Filesystem Hierarchy Standard requirements for a root filesystem.
- We have learned how to use Internet resources to locate and download the source code needed to build a GNU/Linux system.
- We have written basic system startup and shutdown scripts and configured them to execute in the proper runlevels.
- We have included support for multiple users on virtual consoles and implemented permissions on system files.
- But most importantly, we have learned some good design techniques and project management skills that will enable us to tackle any future projects with ease and confidence.

9.2. Planning Next Steps

The Pocket Linux system is nearly overflowing, so there really is no more room to expand the current root diskette to support any additional commands and features. This leaves us with a few choices of where to go next. We can:

- Use the techniques we have learned to design and build an entire GNU/Linux system and install it on a more spacious hard disk partition.
- Remove multi−user capability and some of the less often used commands from the root disk, replacing them with utilities like tar and gzip that would be useful for a rescue/restore diskset.
- Find a way to expand the current system just enough to host a small application. (For more information about hosting applications with Pocket Linux, see Appendix A)

Which ever path is chosen, we can move forward confidently, armed with the knowledge we need to be successful in our endeavors.

Appendix A. Hosting Applications

A.1. Analysis

An operating system by itself is not much fun. What makes an OS great is the applications that can be run on top of it. Unfortunately, the Pocket Distribution currently does not have much room for anything other than system programs. Still, it would be nice to expand the system just enough to host some cool applications. Obviously a full−blown X−Windows GUI is out of the question, but running a small console based program should be within our reach.

Rather than doing a typical "hello world" program as an example, application hosting will be demonstrated using a console based audio player called mp3blaster. Building mp3blaster offers more technical challenge than "hello world" and the finished product should be a lot more fun. However, it should not be construed that a console−based jukebox is the only application for Pocket Linux. On the contrary, after completing this phase the reader should have the knowledge and tools to build almost any console−based program he or she desires.

So what will it take to turn a pocket−sized GNU/Linux system into a pocket−sized mp3 player? A few things are listed below.

- Add support for audio hardware.
- Create space for the mp3blaster program.
- Provide a convenient way to access audio files.

A.2. Design

A.2.1. Support for audio hardware

There is a vast proliferation of audio hardware on the market and each sound card has its own particular configuration. For details on how to set up a particular sound card we can turn to the Sound−HOWTO available from The Linux Documentation Project at http://www.tldp.org. In a broader sense, however, we can treat a sound card like any other piece of new hardware. To add new hardware to a GNU/Linux system we will need configure the kernel to recognize it and configure `/dev` files on the root disk to access it.

A.2.1.1. Kernel support for audio

In order to support sound cards, a new kernel will have to be built. It is very important that audio hardware support be configured as built−in, because the Pocket Distribution is not set up to handle kernel modules.

A.2.1.2. Root disk support for audio

Searching `devices.txt` for the keyword "sound" will list quite a few possible audio devices, but usually only `/dev/dsp` and `/dev/mixer` are required to get sound from a PC. These two files control the digital audio output and mixer controls, respectively.

A.2.2. Creating space for the program

Probably the easiest way to create more space for the mp3blaster program is to mount an additional storage device. There are several choices for mount points. So far `/usr`, `/home` and `/opt` are all empty directories and any one of them could be used to mount a floppy, CD−ROM or additional compressed ramdisk image. The `/usr` directory is a logical choice for a place to put an application, but what about the choice of media? Mp3blaster and its required libraries are too big to fit on a 1.44M floppy and burning a CD−ROM seems like a lot of work for one little program. So given these constraints, the best choice would be to put the program on a compressed floppy.

A.2.2.1. Mounting additional compressed floppies

Mounting CD's and uncompressed diskettes is easy, but what about loading compressed images from floppy into ramdisk? It will have to be done manually, because automatic mounting of compressed floppies only works for the root diskette. And using **mount /dev/fd0** will not work because there is no filesystem on the diskette, there are only the contents of a gzip file. The actual filesystem is contained inside the gzip file. So how can we mount the filesystem buried beneath the gzip file? This puzzle can be solved by examining at the steps used to create the familiar compressed root disk floppy.

1. A ramdisk is created, mounted and filled with files.
2. The ramdisk device is dismounted.
3. The contents of the ramdisk are dumped to an image file using **dd**.
4. The image file is compressed with **gzip**.
5. The compressed image file is written to floppy with **dd**.

If that is how the compressed image makes its way from ramdisk to compressed floppy, then going from compressed floppy to ramdisk should be as simple as running through the steps in reverse.

1. The compressed image file is read from floppy with **dd**.
2. The image file is uncompressed with **gunzip**.
3. The contents of the image file are dumped into ramdisk using **dd**.
4. The ramdisk device is mounted.
5. The files are available.

We can cut out the intermediate image file by using a pipe to combine **dd** and **gunzip** like this: **dd if=/dev/fd0 | gunzip −cq > /dev/ram1**. Now the compressed floppy goes straight into ramdisk, decompressing on the fly.

A.2.2.2. Root disk support for additional ramdisks

We already have kernel support for ramdisks, because we are using a compressed root disk, but we will need to create more ramdisks in `/dev`. Typically the kernel supports eight ramdisks on `/dev/ram0` through `/dev/ram7` with `ram0` being used for the rootdisk. The `devices.txt` file included in the Linux source code documentation will be helpful for matching devices to their major and minor numbers.

A.2.3. Accessing audio files

The sample mp3 file that we will be using in our example is small enough to fit on an uncompressed floppy disk so that there is no need to burn a CD. However, serious music lovers may want to have the capability to mount a custom CD−ROM full of tunes and that option will require support for additional hardware.

A.2.3.1. CD−ROM hardware support

Most modern CD−ROM drives will use IDE devices like `/dev/hdc` or `/dev/hdd`. To support these CD−ROM drives we will have to configure IDE support in the kernel and create the appropriate device files on the root disk.

A.2.3.2. CD−ROM filesystem support

CD−ROM's have different filesystems than hard disks and floppies. Most CD burning applications use a filesystem called ISO−9660 and have the capability to support joliet or rockridge extensions. We will have to include support for these filesystems in the kernel in order to mount CD−ROM's.

A.2.4. Other required files

We will want to have all of the required libraries and other supporting files available as part of the compressed `/usr` image so that mp3blaster can run correctly. The familiar **ldd** command can be used to determine which libraries mp3blaster requires. Any additional libraries can be placed in `/usr/lib`. Even though some of the libraries may appear in `/lib` on the development system, they can still go in `/usr/lib` on the Pocket Linux system. The linker is smart enough to look in both places when loading libraries.

Because mp3blaster uses the curses (or ncurses) screen control library there is one additional file we need. The curses library needs to know the characteristics of the terminal it is controlling and it gets that information from the terminfo database. The terminfo database consists of all the files under the `/usr/share/terminfo` directory and is very large compared to our available disk space. But, since Pocket Linux only supports the PC console, we only have one terminal type to worry about and therefore need only one file. The piece of the terminfo database we need is the file `/usr/share/terminfo/l/linux`, because we are using a "Linux" terminal. For more information about the subject of curses, see John Strang's book entitled "Programming with Curses" available from O'Reilly publishing at http://www.oreilly.com.

A.2.5. Summary of tasks

Between sound cards, ramdisks, CD−ROM's and terminfo there is quite a bit to keep track of. So let's take a moment to organize and summarize the tasks necessary to make the pocket jukebox a reality.

- Create a new kernel disk that includes built−in support for audio hardware, IDE devices and CD−ROM filesystems.
- Create the appropriate `/dev` files on the root disk to support audio hardware, additional ramdisks and IDE CD−ROM's.
- Create a startup script to load a compressed image from floppy into a ramdisk and mount the ramdisk on `/usr`.
- Create a compressed floppy that holds the mp3blaster program, its required libraries and terminfo files.

A.3. Construction

A.3.1. Create an enhanced boot disk

A.3.1.1. Build a new kernel

```
bash# cd /usr/src/linux
bash# make menuconfig
```

Be sure to configure support for the following:

- 386 processor
- Floppy disk
- RAM disk
- Virtual console
- Audio hardware
- CD−ROM hardware
- ISO−9660 and Joliet filesystems

```
bash# make dep
bash# make clean
bash# make bzImage
```

A.3.1.2. Copy the kernel to diskette

Place the boot disk in drive fd0

```
bash# cp /usr/src/linux/arch/i386/boot/bzImage /mnt/boot/vmlinuz
bash# mount /dev/fd0 /mnt
```

A.3.1.3. Install the LILO boot loader

```
bash# lilo −r /mnt
```

A.3.1.4. Unmount the boot disk

```
bash# cd /
bash# umount /mnt
bash# sync
```

A.3.2. Create an enhanced root disk

A.3.2.1. Create additional device files

A.3.2.1.1. IDE CD−ROM

```
bash# mknod −m640 ~/staging/dev/hdc b 22 0
bash# mknod −m640 ~/staging/dev/hdd b 22 64
```

Optionally create additional IDE devices.

A.3.2.1.2. Ramdisk

```
bash# mknod −m 640 ~/staging/dev/ram1 b 1 1
bash# mknod −m 640 ~/staging/dev/ram2 b 1 2
bash# mknod −m 640 ~/staging/dev/ram3 b 1 3
bash# mknod −m 640 ~/staging/dev/ram4 b 1 4
bash# mknod −m 640 ~/staging/dev/ram5 b 1 5
bash# mknod −m 640 ~/staging/dev/ram6 b 1 6
bash# mknod −m 640 ~/staging/dev/ram7 b 1 7
```

A.3.2.1.3. Audio

```
bash# mknod −m664 ~/staging/dev/dsp c 14 3
bash# mknod −m664 ~/staging/dev/mixer c 14 0
```

A.3.2.2. Write a startup script to mount a compressed floppy

Use a text editor to create the following script and save it as `~/staging/etc/init.d/usr_image`

```
#!/bin/sh
#
# usr_image − load compressed images from floppy into ramdisk and
#             mount on /usr.
#
echo −n "Is there a compressed diskette to load for /usr [y/N]? "
read REPLY
if [ "$REPLY" = "y" ] || [ "$REPLY" = "Y" ]; then
  echo −n "Please insert the /usr floppy into fd0 and press <ENTER>."
  read REPLY
  echo "Clearing /dev/ram1."
  dd if=/dev/zero of=/dev/ram1 bs=1k count=4096
  echo "Loading compressed image from /dev/fd0 into /dev/ram1..."
  (dd if=/dev/fd0 bs=1k | gunzip −cq) >/dev/ram1 2>/dev/null
  fsck −fp /dev/ram1
  if [ $(($?)) −gt $((1)) ]; then
    echo "Filesystem errors on /dev/ram1!  Manual intervention required."
  else
    echo "Mounting /usr."
    mount /dev/ram1 /usr
  fi
fi
#
# end of usr_image
```

Configure the script to run right after root is mounted.

```
bash# ln −s ../init.d/usr_image ~/staging/etc/rcS.d/S21usr_image
```

A.3.2.3. Create a compressed root disk

```
bash# cd /
bash# dd if=/dev/zero of=/dev/ram7 bs=1k count=4096
bash# mke2fs −m0 /dev/ram7
bash# mount /dev/ram7 /mnt
bash# cp −dpR ~/staging/* /mnt
bash# umount /dev/ram7
bash# dd if=/dev/ram7 of=~/phase8−image bs=1k
```

```
bash# gzip -9 ~/phase8-image
```

Insert the diskette labled "root disk" into drive fd0.

```
bash# dd if=~/phase8-image.gz of=/dev/fd0 bs=1k
```

A.3.2.4. Unmount the root disk

```
bash# cd /
bash# umount /mnt
bash# sync
```

A.3.3. Create a compressed /usr disk for mp3blaster

The compressed /usr diskette will be created in using the same process that is used to create the compressed root disk. We will copy files to a staging area, copy the staging area to ramdisk, compress the ramdisk and write it to diskette.

A.3.3.1. Create a staging area

```
bash# mkdir ~/usr-staging
bash# cd ~/usr-staging
bash# mkdir bin lib
bash# mkdir -p share/termcap/l
```

A.3.3.2. Install the mp3blaster program

Download the latest version of mp3blaster source code from its home at http://www.stack.nl/~brama/mp3blaster.

```
bash# cd ~/usr/src/mp3blaster-3.13
bash# ./configure
bash# make
bash# cp src/mp3blaster ~/usr-staging/bin
```

A.3.3.3. Copy additional libraries and terminfo

Note: This is an example from the author's development system. Different systems may yield slightly different results.

```
bash# cd ~/usr-staging/lib
bash# ldd ~/usr-staging/bin/mp3blaster
bash# cp /usr/lib/ncurses.so.5.0  .
bash# cp /usr/lib/stdc++.so.3 .
bash# cp /lib/libm.so.6 .
bash# cp /usr/lib/libgcc_s.so.1 .
bash# cd ~/usr/staging/share/terminfo/l
bash# cp /usr/share/terminfo/l/linux .
```

A.3.3.4. Make a compressed image and copy it to diskette

```
bash# cd /
bash# dd if=/dev/zero of=/dev/ram7 bs=1k count=4096
bash# mke2fs -m0 /dev/ram7
bash# mount /dev/ram7 /mnt
bash# cp -dpR ~/usr-staging/* /mnt
bash# umount /dev/ram7
bash# dd if=/dev/ram7 of=~/mp3blaster-image bs=1k
bash# gzip -9 ~/mp3blaster-image
```

Insert the diskette labled "mp3blaster" into drive fd0.

```
bash# dd if=~/mp3blaster-image.gz of=/dev/fd0 bs=1k
```

A.3.4. Create a data diskette for testing

Go to the internet site http://www.paul.sladen.org and download the mp3 file of Linus Torvalds pronouncing "Linux." The direct link is: http://www.paul.sladen.org/pronunciation/torvalds-says-linux.mp3. Create a Second Extended (ext2) filesystem on a floppy and copy the mp3 file onto the diskette.

A.4. Implementation

A.4.1. System Startup

1. Boot from the kernel diskette.
2. Insert the root floppy when prompted.
3. When prompted for a /usr diskette, say 'Y'.
4. Insert the mp3blaster diskette and press **Enter**.

A.4.2. Verify that the /usr diskette loaded properly

```
bash# mount
bash# ls -lR /usr
```

A.4.3. Check the audio device initialization

```
bash# dmesg | more
```

If everything worked there should be a line or two indicating that the kernel found the audio hardware. The example below shows how the kernel might report a Yamaha integrated sound system.

```
ymfpci: YMF740C at 0xf4000000 IRQ 10
ac97_codec: AC97 Audio codec, id: 0x4144:0x5303 (Analog Devices AD1819)
```

A.4.4. Test audio output

```
bash# echo "10101010" > /dev/dsp
```

A short burst of static coming from the PC speakers indicates that sound is working.

A.4.5. Play a sample file

```
mount /dev/fd0 /home
bash# /usr/bin/mp3blaster
```

Use mp3blaster to select and play the file `/home/torvalds-says-linux.mp3`. Use mp3blaster's mixer controls to adjust the volume as needed.

A.4.6. System shutdown

Bring the system down gracefully with the **shutdown** command.